Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Implementing Delay Lines on SHARC® Processors

Contributed by Divya Sunkara

Rev 1 – October 25, 2006

Introduction

This EE-Note discusses the implementation of a delay line on the ADSP-2126x, ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, and ADSP-21366 SHARC® series of processors, hereafter representatively referred to as ADSP-21262 processors.

Because ADSP-21262 processors access external memory via the parallel port, special considerations are required while implementing delay lines on these processors when compared to other SHARC processors.

Discrete filters can be implemented with sample-based delay lines and a set of coefficients that define the filter. Due to memory constraints, delay lines are usually implemented in external memory using various techniques.

This document discusses two delay-line implementations using the parallel port. First, a sample-based delay line using parallel port core access is discussed. Later in the document, techniques used in a block-based delay line implemented via parallel port DMA are described.

Sample-Based Delay Lines

Delay effects are achieved by implementing finite impulse response (FIR) or infinite impulse response (IIR) filters. For sample-based delay lines, this document focuses on the implementation of a simple stereo delay effect algorithm with cross feedback. The left and right channels are coupled by introducing cross-feedback coefficients, so that the delayed output of one channel is fed into the input of the other channel, creating a stereo delay effect. For details, refer to the sample-based delay line code for the ADSP-21262 processor located in the associated `.ZIP` file.

The implementation of the above algorithm requires two delay lines: one for left channel, and one for the right channel. The delay lines are implemented as circular buffers in external memory. The length of the delay line depends on the required delay and the input sampling rate. For example, a 0.5-second delay requires a 24000 tap delay line for an 48-KHz input sampling rate. Since this implementation requires feedback from the delayed output, the total delay length is 24001, which includes the previous sample for the delay. Since the external memory is 8 bits wide, each tap (32-bit word) requires four external memory locations. Hence, the total delay-line buffer length is four times the number of taps.

Due to its feedback structure, this algorithm requires a read from the delay-line buffer followed by a write. The read and write occur in a sample-based manner. Because the core accesses external memory via parallel port DMA registers (`EIPP`, `EMPP`, and `ECPP`), the circular buffers in external memory are implemented by using the DAG registers. Even though the DAG registers cannot access external memory, the DAG registers' circular address capabilities are used to calculate the address to access the external memory via the parallel port. The DAG register pointing to external circular buffers are then assigned to the parallel port external index register (`EIPP`) in order for the core to access the data. Initially, the read pointer points to the buffer's last tap and the write pointer points to the first tap. Following each read and write, both pointers are decremented and the transfers continue sample by sample. For sample-based processing, a single output is transmitted at a particular time.

Core-driven transfers to the external memory via the parallel port can be initiated using four techniques as described in the *ADSP-2126x SHARC Processors Peripherals Manual* as well as in the *ADSP-2136x SHARC Processor Hardware Reference*. The known-duration access technique, which is the most efficient of all the accesses, is used for reads and writes. With 8-bit external memory, each 32-bit access via the parallel port requires a known duration of 15 cycles [`(1 ALE-cycle * 3 CCLK) + (4 data-cycles * 3 CCLK)`] with a `PPDUR` (parallel port duration setting) of 3. The program can use the 15 cycles that occur after the data is written to the transmit register (or read from the receive register of parallel port) efficiently for other purposes.

Another consideration while accessing data via the parallel port is the addressing of external memory. This is due to the fact that the parallel port no longer accepts logical addresses but requires physical addresses of the external memory to transfer data. Consecutive 32-bit values in external memory are addressed by an offset of 4 because each 32-bit value occupies four-byte-wide locations of external memory.

Block-Based Delay Lines

For the block-based delay-line technique, this document discusses the implementation of a low-reflection setting FIR as shown in Figure 1.

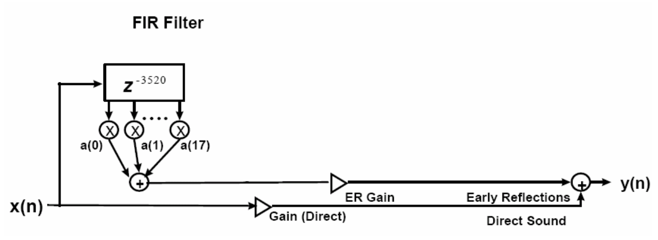

Figure 1. FIR Filter

The impulse response of the filter is available in Figure 2. Figure 3, which illustrates the FIR implementation using block-based delay lines in external memory, depicts the implementation with a block of eight samples, a filter length of six, and two tap accesses (tap2 and tap4). Figure 1 shows the FIR implementation of a single block of data.

The FIR implementation of a single block of input data requires three processes: a write DMA, read DMAs, and the delay process (convolution of input and coefficient values). Each of these processes is discussed in detail below.

Even though the block-based delay line code implements the filter shown in Figure 1 and Figure 2, Figure 3 shows the implementation of only two taps of the filter to simplify the illustration; the additional taps follow the same template.

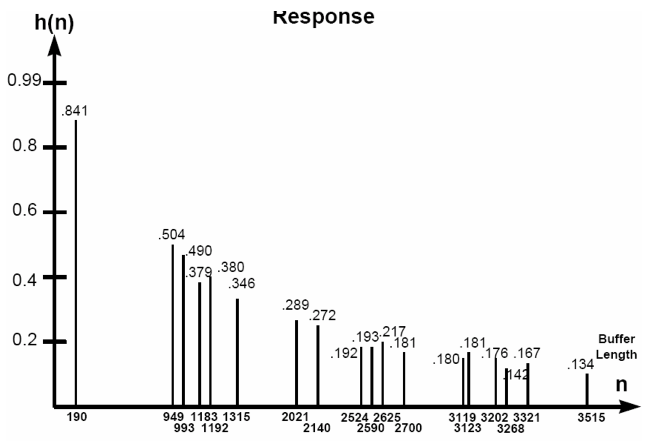

Figure 2. FIR Filter Response

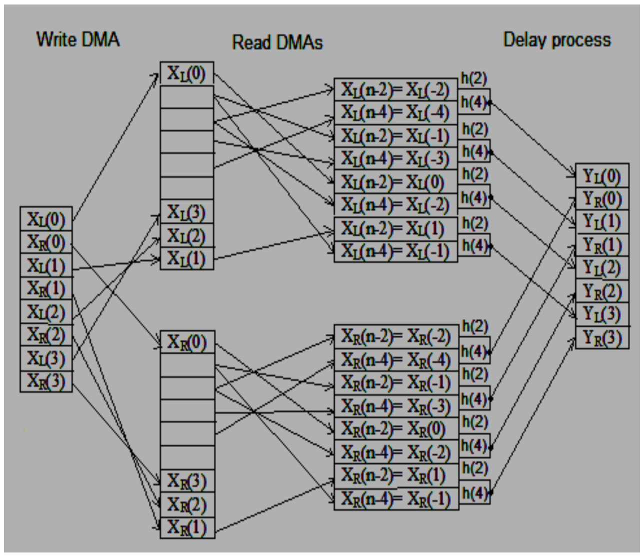

Figure 3. FIR Filter Implementation

Write DMA

The sample data arrives as a block of `NUM_SAMPLES`. The data in the input buffer contains the left and right channel samples as shown in Figure 4. Therefore, two delay lines are required for each of the channel. The processing on each of the channels is identical. The length of each delay line is equal to:

```
Filter Length + ((NUM_SAMPLES/2) -1)
```

For block-based FIR implementations, the processing starts with writing a block of data (write DMA) to the delay-line buffer in external memory. Figure 4 illustrates the write DMA process to the left and right delay lines. With a block length (`NUM_SAMPLES`) of eight and a filter length of six, the delay length of each of the channels is nine. Each of the nine locations of the delay line shown in Figure 4 corresponds to four locations in external byte-wide memory. Therefore, the total delay line length in external memory is 28 (9 x 4). The write pointers for each of the channels are initialized to the first locations of the left and right delay line buffers shown as `WrLptr` and `WrRptr` in Figure 4. Two write DMAs take place, one for each channel. The function used to perform a write DMA for both channels is the same (Listing 1). The parameters passed to this function vary, and are based on a particular channel. The DMA for the left channel takes place with the internal index register pointing to `TrLptr` and the external index register of parallel port pointing to `WrLptr`. Similarly, for the right channel, the internal index register points to `TrRptr` and external index register points to `WrRptr`. The internal modify register is set to 2 and the external modify register is set to -1, placing each channel data in counter clockwise direction in external delay line memory starting with the first sample located at the start of the delay line buffer as shown in Figure 4.

```
void TransDMA(int  *Tinptr,  int Tcount, int
*Textptr)
{
     *pPPCTL = 0;

/* initiate parallel port DMA registers for
first DMA*/
     *pIIPP = Tinptr;
     *pIMPP = 2;
     *pICPP = Tcount;

     *pEMPP = -1;
     *pEIPP = Textptr;
/* For 8-bit external memory, the External
count is four times the internal count */
     *pECPP = Tcount * 4;

     *pPPCTL = PPTRAN| PPBHC| PPDUR4;
     *pPPCTL |= PDEN|PPEN; // enalble parallel
port DMA

/*poll to ensure parallel port has completed
the transfer*/
     do{
     ;}
     while( (*pPPCTL & (PPBS|PPDS) ) != 0);

     *pPPCTL = 0; // disable parallel port
     // start of second DMA
}
```

Listing 1. TransmitDMA.c

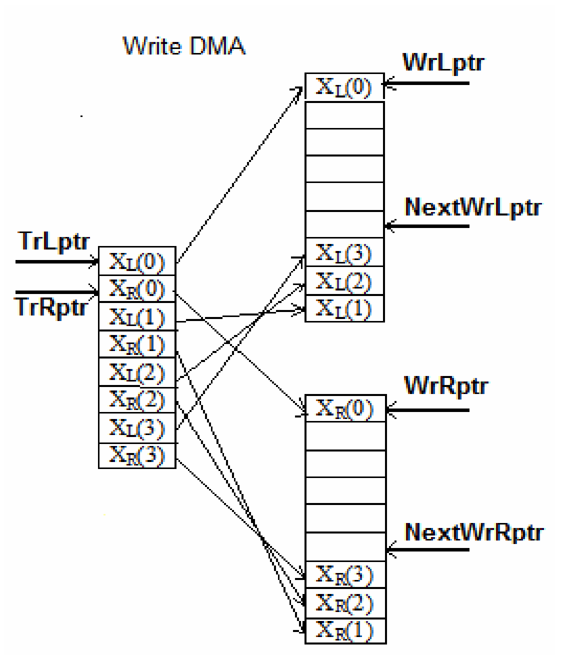

Figure 4. Write DMA

Circular Buffering

Since the parallel port DMA parameter registers do not take into account the wrapping of circular buffers, if the write pointer needs to wrap around the end, the DMA is split into two separate DMA sections. One of the DMA sections starts at the write pointer and the other DMA section begins at the end of circular delay line buffer for that channel. For example, if the DMA for left channel starts at `WrLptr` (see Figure 4), it would be split into two DMAs with one starting at `WrLptr` and transferring one word (4 bytes), and the other starting at the end of the left channel delay line buffer, transferring four words (16 bytes).

The function that tests for this wrapping appears in Listing 2. In the function, the `TestDMA` variable points to `Num_Sample/2` locations (the number of samples transferred in a particular channel per DMA) offset from the start of external delay line of a particular channel.

```
void WriteDMA( int *Inptr,  int *Wrptr, int
*TestDMA, int *BUF)
{
int DMAcount1,DMAcount2,Inc;
     int*Wrptr1,*Wrptr2,*Inptr1,*Inptr2;

     if (Wrptr < TestDMA)
     {
     DMAcount2 = (TestDMA-Wrptr)/4;
     DMAcount1 = ((NUM_SAMPLES)/2) - DMAcount2;
     Wrptr1 = Wrptr; // Pointer to the start DMA
     Inc = -(DMAcount1 * 4);
     Wrptr2 = Wrptr1; // external pointer for
first DMA
     Wrptr2 = circptr(Wrptr2, Inc, BUF, LENGTH);
// external pointer for second DMA
     Inptr1 = Inptr; // internal pointer for
first DMA
     Inptr2 = Inptr1+ DMAcount1*2; // internal
pointer for second DMA

     TransDMA(Inptr1, DMAcount1, Wrptr1);
     TransDMA(Inptr2, DMAcount2, Wrptr2);
     }

     else
     {
     TransDMA(Inptr, NUM_SAMPLES/2, Wrptr);
     }

     RdptrInit= Wrptr; // Next Read pointer for
read DMA
     Inc =-((NUM_SAMPLES/2)*4);
     WrptrTemp = Wrptr;

WrptrTemp = circptr(WrptrTemp,Inc,BUF,LENGTH);
// Next write pointer for write DMA
}
```

Listing 2. writeDMA.c

Read DMA

For each write DMA to the external delay line buffers, the number of read DMAs correspond to the number of taps accessed in the FIR filter. Figure 5 depicts the read DMA from the left and right delay line buffers. As described above, two taps are accessed in this illustration (`tap2` and `tap4`), and there are two read DMAs for every write DMA. The length of the internal buffer to perform these read DMAs is calculated by multiplying the number of samples transferred per DMA times the number of taps accessed. With regard to Figure 5, the internal buffer length for read DMAs of each of the channels is eight [`4 (number of samples per DMA) * 2 (number of taps accessed)`]. The function that performs the DMA from the delay line buffer to the internal buffer is given in Listing 3.

The DMA takes place with the internal modify value set to number of taps and the external modify value set to -1. As shown in Figure 5, two read DMAs are performed for each of the channels with the external index register pointing to `RdLptr1` and `RdLptr2` for the left channel (and `RdRptr1` and `Rdrptr2` for the right channel), and the corresponding internal index register points to `ReLptr1` and `ReLptr2` for left channel (and `RdRptr1` and `RdRptr2` for right channel). Each sample transferred from external memory requires a corresponding location in internal memory, based on the modify register values as shown in Figure 5.

`RdLptr` and `RdRptr` act as the reference pointers, obtaining the external index register values for read DMAs for each of the channels; they are always initialized to the current write DMA pointers (`WrLptr` and `WrRptr`) of each of the channels, respectively. The read pointer for each of the taps is derived by offsetting the tap number from the reference. The internal index for each successive read DMAs is incremented by one. The function that sets the external and internal index pointers for each of the read DMA is shown in Listing 4. This function also checks for wrapping of circular buffers similar to that discussed in the Write DMA section of this document. The reference pointer for the read DMAs of the left and right channels for next set of data is set to the write pointers for the next set of data, which are `NextWrLptr` (left channel write pointer) and `NextWrRptr` (right channel write pointer) with respect to Figure 4.

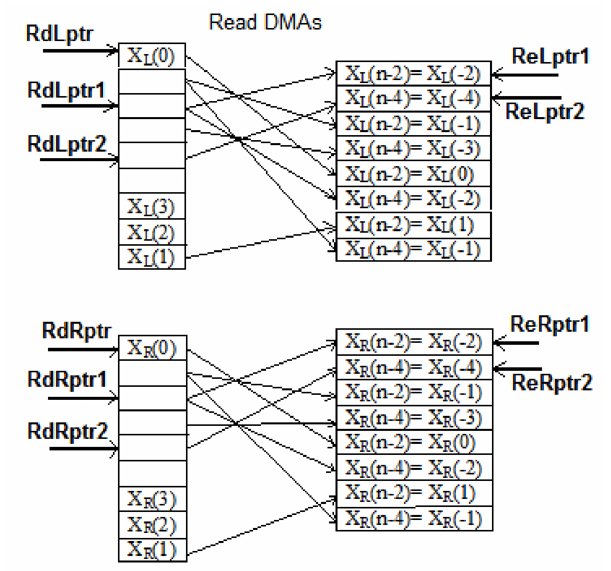

Figure 5. Read DMAs

```
void RecivDMA(int *Rinptr, int Rcount, int
*Rextptr)
{

     *pPPCTL = 0;

     *pIIPP = Rinptr;
     *pIMPP = NUM_TAPS;
     *pICPP = Rcount;

     *pEMPP = -1;
     *pEIPP = Rextptr;
/* For 8-bit external memory, the External
count is four times the internal count */
     *pECPP = Rcount * 4;

     *pPPCTL = PPBHC|PPDUR4; /* Receive(Read) */

     *pPPCTL |= PPDEN|PPEN; // Enable parallel
port DMA

/*poll to ensure parallel port has completed
the transfer*/
     do{
     ;}
     while( (*pPPCTL & (PPBS|PPDS) ) != 0);

     *pPPCTL = 0; // disable parallel port

}
```

Listing 3. ReceiveDMA.c

```
void ReadDMA(int *Inptr, int *Rdptr, int
*TestDMA, int *BUF)
{

int DMAcount1,DMAcount2,Inc;
int *Rdptr1,*Rdptr2,*Inptr1,*Inptr2;

int *RdptrTemp,i,InptrTemp;

int Tapno[NUM_TAPS] =
{993,1315,2524,2700,3202,2700,3119,3123,3202,32
68,3321,3515};

InptrTemp = Inptr;

for(i=0; i<NUM_TAPS; i++)
{
     RdptrTemp = Rdptr;
     Inc = Tapno[i] *4;
     RdptrTemp = circptr(RdptrTemp, Inc,BUF,
LENGTH);

     if (RdptrTemp < TestDMA)
     {

           DMAcount2 = (TestDMA-RdptrTemp)/4;
           DMAcount1 = ((NUM SAMPLES)/2) -
DMAcount2;
           Rdptr1 = RdptrTemp;
           Inc = -(DMAcount1 *4);
           Rdptr2 = Rdptr1;
           Rdptr2 = circptr(Rdptr2, Inc, BUF,
LENGTH);
           Inptr1 = InptrTemp;
           Inptr2 = Inptr1+ DMAcount1*NUM_TAPS;
// internal pointer for second DMA

           RecivDMA(Inptr1, DMAcount1, Rdptr1);
           RecivDMA(Inptr2, DMAcount2, Rdptr2);

     }
    else
     {
           RecivDMA(InptrTemp, NUM_SAMPLES/2,
RdptrTemp);
     }

InptrTemp = Inptr+1;

}

}
```

Listing 4. ReadDMA.c

Delay Process

After the read DMAs for all the required taps are completed, processing takes place on the values read into the internal memory.

Figure 6 shows the delay process on the internal memory data. This process is identical for both channels. The tap values for each of the channels are in consecutive locations in internal memory as a result of the read DMA. The delay process implements an FIR filter, which involves multiplying the tap data value times its corresponding coefficient values and adding the results to get the sample output. Consecutive samples are calculated in the same manner for both the left and right channels. The processed left and right samples are placed in alternate locations to be transmitted out in I2S mode.

```
DelayProcess(int*InBlockptr,int*InBufptr)
{
     int *TempInBufptr;
     float EarlyRef, InSample;
     int x,j,m,i,k,l;
 float
TapValue[NUM TAPS]={0.490,0.346,0.192,0.181,0.1
76,0.181,0.180,0.181,0.176, 0.142,0.167,0.134};

     for (k=0;k<(NUM_SAMPLES/2);k++)
     {
           TempInBufptr = InBufptr;
           j = k*2;  // increment for block
pointer
           m = k*NUM TAPS;// increment for
Buffer pointer
           InSample = *(InBlockptr + j)*0.33 ;
           EarlyRef = 0.0;
           for (i=0; i<NUM_TAPS; i++)
           {
                 x=m+i;
                 EarlyRef = EarlyRef +
(*(TempInBufptr+x) *  TapValue[i]);

                 }

                 *(InBlockptr + j) =
(int)(InSample + EarlyRef * 0.33);

}
```

Listing 5. Delayprocessing.c

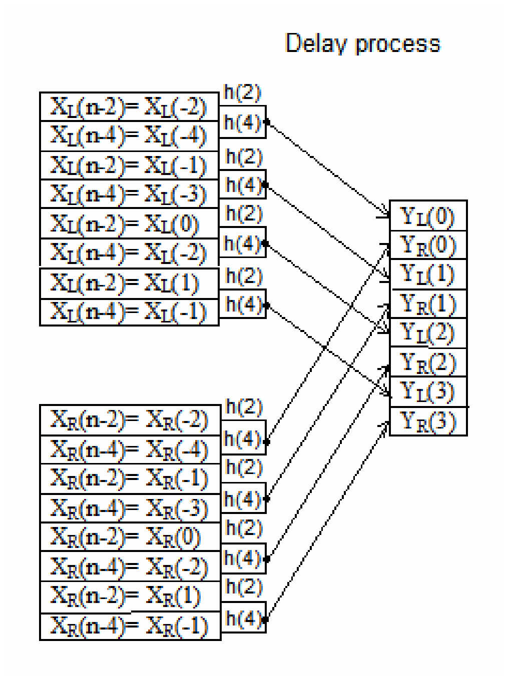

Figure 6. Delay Process

Conclusion

This document discusses the methods used to implement delay lines for parallel port-based processors such as the ADSP-2126x, ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365 and ADSP-21366 SHARC series of processors. Implementation methods for sample-based and block-based delay line are described. The program code for each of these techniques for ADSP-21262 processors is available in the associated `.ZIP` file.

For an FIR filter implementation with a block-based delay line, an input block of 512 samples with a filter length (delay) of 3520 and 18 tap accesses takes 463,814 core clock cycles at a clock rate of 200 MHz for ADSP-21262. Increasing the number of input block samples or the number of tap accesses increases the processing time for implementing the filter.

References

[1] *ADSP-2126x SHARC® Processor Peripherals Manual*. Revision 3.0, December, 2005. Analog Devices, Inc.

[2] *ADSP-2136x SHARC® Processor Hardware Reference.* Revision 1.0, October 2005. Analog Devices, Inc.

[3] *Introduction to Signal Processing,* S. J. Orfanidis, Prentice-Hall ,1996

[4] *About This Reverberation Business*, J. A. Moorer, *Computer Music Journal*, vol. 3(2), pp.13-18 (1979).

Document History

Revision	**Description**
Rev 1 – October 25, 2006 by D.Sunkara	*Initial version*